# 八皇后问题

## 经典八皇后问题

在国际象棋种，皇后是最强大的一枚棋子，可以吃掉与其在同一行，同一列和斜线上的敌方棋子。

八皇后问题就是将八个皇后摆在一张8\*8的国际象棋棋盘上，使每个皇后都无法吃掉别的皇后，一共有多少种摆法？

这个问题是典型的回溯法解决问题。

来自刘汝佳《算法竞赛入门经典》的算法：

void search(int row){

if(row==n) count++;//递归边界。只要走到了这里，所有皇后必然不冲突。

else for(int i=0;i<n;i++){

C[row]=i;//尝试把第row行的皇后放在第i列。

int ok=0;

for(int j=0;j<row;j++)//检查是否和前面的皇后冲突

if(C[row]==C[j]||row-C[row]==j-C[j];row+C[row]=j+C[j])//判断条件

{ok=0;break; }

If(ok==2) search(row+1);

}

}

因为

因此只要传入皇后的个数，即可算出有多少种摆法。

那么判断条件为什么是这样呢？

因为是按行摆放，所以同行就不用考虑了。

而不能同列的判断条件就是和前面已经存在的行依次做比较：C[row]==C[j]

而不能同斜线的判断条件就是和已经存在的行依次做如下比较：

row-C[row]==j-C[j];row+C[row]=i+C[i]

行和列都比较好理解。关键是斜线的判断条件问什么是这样呢。

其实斜线可以分为主对角线和副对角线。

如果我们把整个棋盘看作是一个作标系。每个点的坐标为（row,col），每一条主对角线的斜率都是1，副对角线的斜率都是-1。所以每一条主对角线的函数公式为，col=row+t。所以每一条主对角线上的格子都满足，col-row=t，从而得出若row-C[row]==j-C[j]这个条件为真，则格子（row,C[row]）必然在一条对角线上;而副对角线上的函数方程为col=-row+t，同上可以推出row+C[row]==j+c[j]的判断条件。

只要新进入的行满足上诉条件，就说明一定不会有问题。否则就回溯到上一个条件，选择下一个可能存放的位置。

以后需到这种在固定区域的判断问题，可以尝试利用建立坐标系，然后建立函数公式，再来计算条件的方式来解决问题。

## 经典八皇后问题的扩展（1）

若已知八皇后问题的解是92。

输入：

第一行是一个整数n,代表接下来要输入的整数个数，1<=n<=92。

接下来n行每行一个整数k，代表了把八皇后问题的第k个解。（解的形式是一个8位整数，从左到右分别为第一行到第八行的列序号。根据这些列序号组成的8位整数的大小来排序）。

输出：

输出n行，每行一个整数，和输入的n个数一一对应，以8位整数的形式输出输入的数代表的解。

解题思路：求出8皇后问题的所有8位数解，存放在一个list中，然后调用Collections的sort方法排序这个list。再遍历输入数组，依次输出对应的解。解题代码如下：

**package** shunfeng;

**import** java.util.ArrayList;

//

**import** java.util.Collections;

**import** java.util.Scanner;

//八皇后的问题

**public** **class** EngihtQueens {

**static** **int** *b*[]=**new** **int**[9];

**static** **int** *count*=1;

**public** **static** **void** main(String[] args) {

Scanner in = **new** Scanner(System.***in***);

**int** n=in.nextInt();

ArrayList<Integer> list=**new** ArrayList<Integer>();

**int** a[]=**new** **int**[n];

**for**(**int** i=0;i<n;i++){

**int** temp=in.nextInt();

a[i]=temp;

}

*queen*(0,list,0);

Collections.*sort*(list);

**for**(**int** i=0;i<n;i++) {

System.***out***.println(list.get(a[i]-1));

}

}

**public** **static** **void** queen(**int** n,ArrayList<Integer> list, **int** index){

**if**(index==8){

list.add(n);

*count*++;

**return** ;

}

**for**(**int** i=0;i<8;i++){

*b*[index]=i;

**if**(*is\_OK*(index))

*queen*(n\*10+i+1,list,index+1);

}

}

**public** **static** **boolean** is\_OK(**int** index) {

**for**(**int** i=0;i<index;i++) {

**if**(*b*[index]==*b*[i]||index-*b*[index]==i-*b*[i]||index+*b*[index]==i+*b*[i])

**return** **false**;

}

**return** **true**;

}

}

# 两个字符串的编辑距离（Edit distance）

问题描述：

/\*\*

\* Given two words word1 and word2, find the minimum number of steps required to

\* convert word1 to word2. (each operation is counted as 1 step.)

\*

\* You have the following 3 operations permitted on a word:

\*

\* a) Insert a character

\* b) Delete a character

\* c) Replace a character

\*/

有两个字符串word1和word2,找到把word1转化为word2需要的最小步骤数。你可以使用如下三种操作：

1. 插入一个元素
2. 删除一个元素
3. 代替一个元素

这是一个很明显的动态规划的问题，声明一个数组dp[i][j]表示字符串word1[0..i-1]到word2[0…j-1]的距离。

我们首先可以把这个问题分成两种情况，一种是正常的两个非空字符串之间的转换，一种是至少有一个空字符串的转换。

我们先看至少有一个空字符串之间的转换，转换成数组，就相当于是求dp[i][0]或dp[0][j]的值。那么很显然只要进行i步或者j步的插入或删除操作就可以了。所以有以下结论：

dp[i][0]=i;

dp[0][j]=j;

接下来我们看第二种情况，也就是正常情况，两个非空字符串的转换。那么根据动态规划的思想，我们可以把这个问题拆分成更小的问题。假设我们已经知道了怎么用最小的步数把word1[0…i-2]转换成word2[0…j-2]，即dp[i-1][j-1]已知。那么我们接下来要来考虑word1[i-1]和word2[j-1]的关系，如果它们两个相等，那么很显然，dp[i][j]=dp[i-1][j-1]。

当它们不相等的时候，我们就需要考虑到以下三种情况：

1. 我们可以用word2[j-1]来替换word1[i-1]的方式把word1转换为word2,那么dp[i][j]=dp[i-1][j-1]+1。
2. 我们可以通过删除word1[i-1]的方式来把word1,转换成word2。这时word1[0…i-2]必须要是等于word2[0…i-1]。假设我们已经知道了如何把word1[0…i-2]转换成word2[0…i-2],即dp[i-1][j]已知。那么dp[i][j]=dp[i-1][j]+1;
3. 我们可以通过往word1[0…i-1]中插入一个元素的方式把word1转换为word2。那么这时必然wrod1[0…i-1]必然是要等于word2[0…i-2]。假设我们已经知道了如何把word1[0…i-1]如何转换为wrod2[0…i-2]，即知道了dp[i][j-1]。那么dp[i][j]=dp[i-1][j-1]。

所以我们要求最小步数，就可以取上面三种情况的最小值。

通过上面的思想，我们可以得到如下代码：

**public** **static** **int** editDistance(String word1,String word2) {

**int** n=word1.length();

**int** m=word2.length();

**int** dp[][]=**new** **int**[n+1][m+1];

**for**(**int** i=0;i<=n;i++)

**for**(**int** j=0;j<=m;j++) {

**if**(i==0) dp[i][j]=j;

**else** **if**(j==0) dp[i][j]=i;

**else** **if**(word1.charAt(i-1)==word2.charAt(j-1)) dp[i][j]=dp[i-1][j-1];

**else** dp[i][j]=Math.*min*(dp[i-1][j-1]+1,Math.*min*(dp[i][j-1]+1, dp[i-1][j]+1));

}

**return** dp[n][m];

}

# 括号问题

最简单的括号问题就是判断一个括号字符串是否是合法的。

这个有很多种解法，时间复杂度是o(n)。我认为比较简单的一种方法是：

设置一个参数left，用来表示当前左括号减去右括号的差值，初始值为0。遍历整个字符串，碰到左括号left++。碰到右括号，left--，并判断left是否小于0，如果小于0，则终止遍历，返回字符串不合法。遍历完成后，判断left是否等于0，不等于0则，字符串不合法，等于0则合法。

代码略。

括号问题的扩展：求合法括号字符串对

若有两个由括号组成的字符串，s1和s2。若s1+s2是一个合法的字符串，则认为s1,s2是一个合法的字符串对，并且若s2+s1也是一个合法的字符串，那么s2,s1也是一个合法的字符串对，与前面的不是同一个。而且若s1+s1为一个合法字符串，那么这也算做一个合法字符串。

那么现在给定输入如下：

第一行，数字n,

接下来n行为n个字符串。

输出：

一个整数，为上面n个字符串种的合法字符串对的个数。

解题思路：

首先，我们知道合法的字符串和合法的字符串相组合必然是合法的字符串。而合法的字符串和不合法的字符串相组合必然是不合法的字符串。

那么我们可以首先遍历一次这个n个字符串，把合法的字串的个数统计出来，假设为为countLegal，那么合法字符串包含的合法字符串对的个数就是countLegal的平方。

而在遍历不合法的字符串的时候，我们可以把中间合法的字符串给剔除掉，剩下的不合法字符串只可能有三种情况，第一种是全部是左括号，第二种是全部是右括号，第三种是先是一串连续的右括号，再是一串连续的左括号。

显然，第三种和任何字符串组合都不可能出现合法的字符串。

只有第一种和第二种组合才有可能出现合法字符串，合法的条件显然就是第一种的左括号个数和第二种的右括号的个数相等。那么我们在遍历的时候只需要把第一种和第二种的括号个数存储下来，再对括号个数进行遍历，就可以得到不合法的字符串种存在多少个合法字符串对。

而消除不合法括号中间的合法括号的方法是，利用栈来遍历字符串，遇到左括号直接入栈，遇到右括号，先判断栈顶元素是否为左括号，如果是就弹出栈顶，如果不是就把右括号入栈。如此，最后剩下的就是消除合法字符串之后的不合法字符串，显然，如果栈为空，那么就说明这个字符串就是一个合法的字符串。

为了方便判断剩下的不合法字符串的内容，我用双端队列来取代了栈。具体代码如下:

**public** **static** **int** BracketCombination(String[] stra,**int** n) {

**int** count=0;

**int** countLegal=0;

ArrayList<Integer> leftList=**new** ArrayList<>();

ArrayList<Integer> rightList=**new** ArrayList<>();

**for**(**int** i=0;i<n;i++) {

Deque<Character> dq=**new** ArrayDeque<>();

**for**(**int** j=0;j<stra[i].length();j++) {

**char** c=stra[i].charAt(j);

**if**(c=='(') dq.push(c);//插入队头

**else** **if**(!dq.isEmpty()&&dq.peek()=='(') dq.pop();//弹出队头

**else** dq.push(c);

}

**if**(dq.isEmpty()) countLegal++;//栈空，说明是合法队列

**else** **if**(dq.peekLast()=='(') leftList.add(dq.size());//如果队尾元素是左括号，那么整个队列必然都是左括号，存储左括号的个数

**else** **if**(dq.peekFirst()==')') rightList.add(dq.size());//如果队尾元素是右括号，并且队头元素也是右括号，那么整个队列比然都是右括号，存储右括号的个数。

}

count+=countLegal\*countLegal;

Collections.*sort*(rightList);

**for**(**int** val:leftList) {

**if**(rightList.contains(val))

count+=rightList.lastIndexOf(val)-rightList.indexOf(val)+1;

}

**return** count;

}

# 邮局选址

题目：

一条直线上有居民点，邮局只能建在居民点上。给定一个有序整型数组arr，每个值表示居民点的一维坐标，再给定一个正数num,表示邮局数量。选择num个居民点建立n个邮局，使得所有的居民点到邮局的总距离最短，返回最短的总距离。

方法一:动态规划

首先解决一个问题，如果在arr[0…j]上只能建立一个邮局，最短总距离是多少？

如果是居民点有奇数个，邮局建在中间那个居民点就可以使总距离最短，如果居民点是偶数个，中点有两个，邮局建在哪个都可以让总距离最短。可以根据这个思路计算上面的问题。

生成规模为N\*N的矩阵W，w[i][j]表示如果再arr[i…j]上之只能建立一个邮局，最短的总距离。w[i][j]的值可以通过如下计算得到：

W[i][j]=w[i][j-1]+arr[j]-a[(i+j)/2]